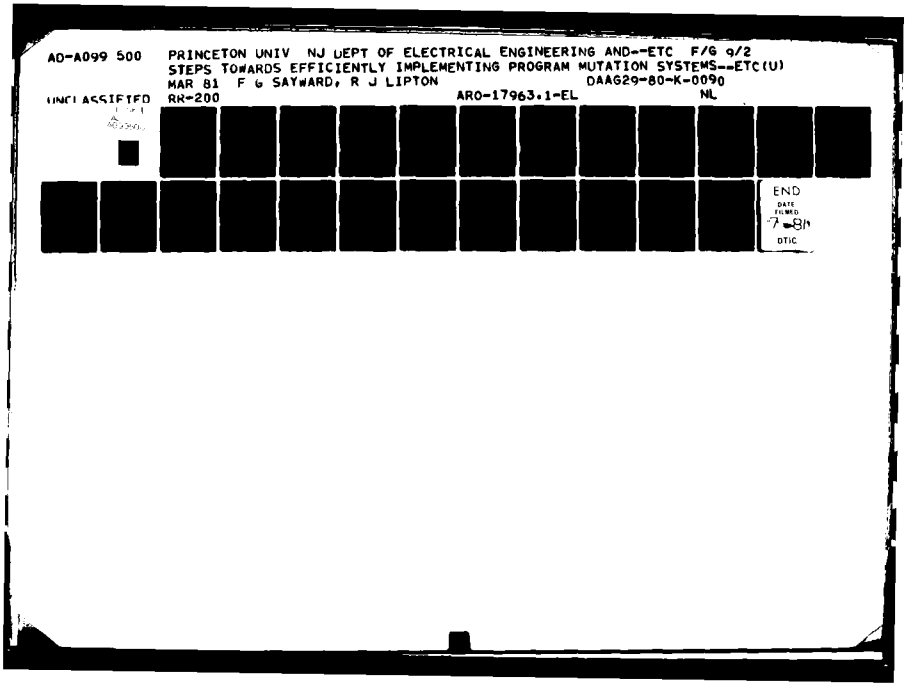
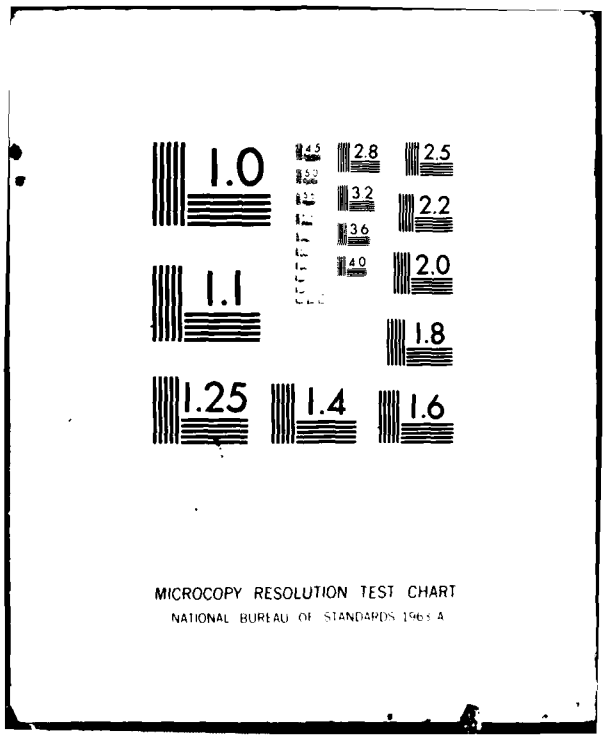

AD-A099 500
PRINCETON UNIV NJ DEPT OF ELECTRICAL ENGINEERING AND--ETC F/G 9/2
STEPS TOWARDS EFFICIENTLY IMPLEMENTING PROGRAM MUTATION SYSTEMS--ETC(U)
MAR 81 F G SAYWARD, R J LIPTON DAAG29-80-K-0090
UNCLASSIFIED RR-200 ARO-17963.1-EL NL
END
DATE FILMED
7-81
DTIC



YALE UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE

II (12)

Steps Towards Efficiently Implementing Program Mutation Systems
The High Level Design of A Distributed Mutation System

Frederick G. Sayward and Richard J. Lipton

Research Report #200

935-3231

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER 500	2. GOVT ACCESSION NO. AD-A099
3. RECIPIENT'S CATALOG NUMBER 500	
4. TITLE (and Subtitle) Steps Towards Efficiently Implementing Program Mutation Systems. The High Level Design of A Distributed Mutation System.	5. TYPE OF REPORT & PERIOD COVERED research
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Frederick G. Sayward and Richard J. Lipton	8. CONTRACT OR GRANT NUMBER(s) DAAG29-80-K-0090
9. PERFORMING ORGANIZATION NAME AND ADDRESS Yale University -- Comp.Sci.Dept. P.O.Box 2158 Yale Sta. New Haven, CT & Princeton Univ.,Elec.Eng.& Comp.Sci.Dept. Princeton, NJ	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Army Institute for Research in Management Information and Computer Science Army Research Office	12. REPORT DATE
	13. NUMBER OF PAGES 17
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this report is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHOR(S) AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE ARMY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Program Testing, Distributed Computing, Networks, Program Mutation, Weak Program Mutation, Compiler Optimization, Mutation Systems, Software Experiments

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Program mutation is a promising method for testing the functional correctness of programs. Its major criticism has been a lack of an efficient implementation method. In this report we will present four methods which have the potential to dramatically improve the time needed to perform program mutation: distributed computation, automatically detecting equivalent mutants, partial mutant execution, and a "weak" version of program mutation.

The presentation of the four potential speed-up methods is given in terms of a high-level design of a prototype distributed program mutation system. One goal

(See reverse side)

400 134

Block # 20.

of this system will be to analyze how the parallelism inherent in program mutation can be exploited on emerging distributed computer systems consisting of many processors. Another goal is to perform experiments to measure the gain in efficiency realizable by a distributed program mutation system and to compare the relative strength of weak program mutation against program mutation. In the report we also indicate areas which should receive special attention in the detailed design of a prototype distributed mutation system if these goals are to be met.

Steps Towards Efficiently Implementing Program Mutation Systems
The High Level Design of A Distributed Mutation System[1]

Frederick G. Sayward
Department of Computer Science
Yale University
New Haven, Connecticut 06520

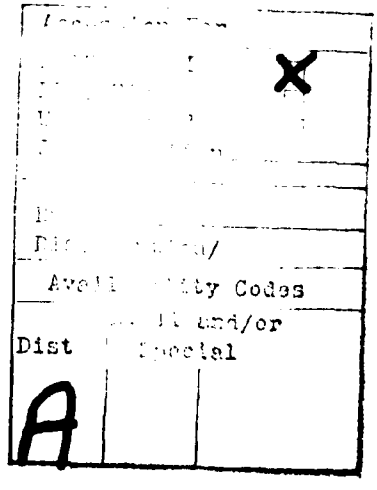
Richard J. Lipton
Department of Electrical Engineering and
Computer Science
Princeton University
Princeton, New Jersey 08540

March 198[illegible]

Abstract

Program mutation is a promising method for testing the functional correctness of programs. Its major criticism has been a lack of an efficient implementation method. In this report we will present four methods which have the potential to dramatically improve the time needed to perform program mutation: distributed computation, automatically detecting equivalent mutants, partial mutant execution, and a "weak" version of program mutation.

The presentation of the four potential speed-up methods is given in terms of a high-level design of a prototype **distributed** program mutation system. One goal of this system will be to analyze how the parallelism inherent in program mutation can be exploited on emerging distributed computer systems consisting of many processors. Another goal is to perform experiments to measure the gain in efficiency realizable by a distributed program mutation system and to compare the relative strength of weak program mutation against program mutation. In the report we also indicate areas which should receive special attention in the detailed design of a prototype distributed mutation system if these goals are to be met.



[1]This research was supported in part by the Army Research Office under Research Contract DAAG29-80-K-0090.

Table of Contents

List of Figures

1. Introduction

One goal of software engineering is to find *efficient* tools which produce from a program *quantitative* life cycle information which has a *well-understood interpretation*. Program mutation [14, 2, 13] is such a tool. Its goal is to provide a measure of how well data T has tested the functional correctness of program P.

In program mutation a set of alternative programs $M_1, M_2, \ldots M_m$, called *mutants*, are constructed from P and run against T. Ideally, each M_i is functionally different from P -- each mutant represents a *potential error* in P. Assuming P runs acceptably on T, M_i failing on T indicates that P does not contain the error represented by M_i. Thus, a quantitative measure of how well P has been tested by T, modulo the represented potential errors, is given by the percentage of failing mutants.

There have been three experimental program mutation systems built, two for testing Fortran programs [6, 5], called FMS.1 and FMS.2, and one for testing Cobol programs [2, 1], called CMS.1. The goal of each system has been to conduct experiments aimed at providing interpretations of the mutant failure percentage -- that is, determining exactly what types of common programming errors can cannot be detected via mutations and comparing the relative strength of program mutation testing to the other contemporary testing methods such as symbolic execution [11, 10, 16] and coverage measures [19, 18, 23]. The results of these experiments have been reported in [22, 2, 7, 1, 8] It is noted that on these systems programs having lengths of up to 1000 statements have been tested under experimental conditions. [15] represents the most comprehensive test by program mutation to date.

All three mutation systems have approached the problem of constructing mutants by defining a set of from 25 to 30 *mutant operators*. A mutant operator is a simple syntactic or semantic program transformation such as changing a particular relational operator to one of the five other relational operators, changing the semantics of a particular Fortran DO loop to act as an Algol FOR loop, or changing a particular variable reference name to be one of the program's other named variables of compatible type. All three mutation systems have implemented only *first-order* mutations which come from a *single* application of a mutant operator on the program P. Analysis has shown that the number of mutants generated by these systems is on the order of N^2 where N is the number of statements in P[2] [22, 8].

The method of mutant operations was chosen because it is conceptually simple and easy to

---

[2] In section 3 it will be explained how the compilation of N^2 mutants is avoided -- only the program P need be compiled

implement. However, using general *program transformations* introduces a new problem -- the generation of mutants which are functionally equivalent to the given program P. These *equivalent mutants* are a nuisance since they add no power to the mutant test and thus must be diligently accounted for during experiments. Furthermore, during some experiments as many as *10%* of the mutants have turned out to be equivalent. A method to automatically detect some types of equivalent mutants, based on compiler optimization techniques, has been designed but it has yet to be implemented [4].

The experiments run on these three prototype program mutation systems have shown that the method will uncover virtually all errors detected by contemporary testing methods and have further shown that program mutation has the potential to detect some errors which are overlooked by all other methods. However, due to the relatively large number of mutants generated it remains to be seen whether or not the method can be implemented efficiently enough to be put into a production environment. This is further aggravated by the above equivalent mutants problem.

2. Potential Speed-Up Methods

All current program mutation systems execute mutants in their entirety on test data. For test data T consisting of n test cases $I_1, I_2, \ldots, I_n$, the program P is first executed to produce corresponding output $O_1, O_2, \ldots, O_n$. In functional notation, $O_i = P(I_i)$. A mutant M_j is said to fail if for any test case i, either M_j has a run-time exception or $M_j(I_i)$ is different from $P(I_i)$. The main loop of current mutation systems is illustrated in figure 2-1. Details will be given in section 3.

In this report we will discuss how the mutation test can potentially be more efficiently implemented by overviewing the design of a new prototype program mutation system which radically differs from all previous systems. Rather than tackle a new language, we will concentrate on performing program mutation on Fortran programs. The following four potential speed-up factors will be incorporated.

2.1. Distributed Computation

Figure 2-1 illustrates the sequential nature of current program mutation systems. However, there is nothing inherently sequential in the method. Indeed, the nature of mutation analysis in which several mutants are independently run against test data suggests that the method is a "natural" for distributed computing. For a distributed system with r processors, one can conceive of routing mutants and test data to free processors with a potential speed-up factor of r. The method is particularly suited to local area networks of personal machines [20].

The detailed design of a prototype *distributed* program mutation system should focus on the communication issues related to realizing as close as possible the potential r speed-up factor afforded

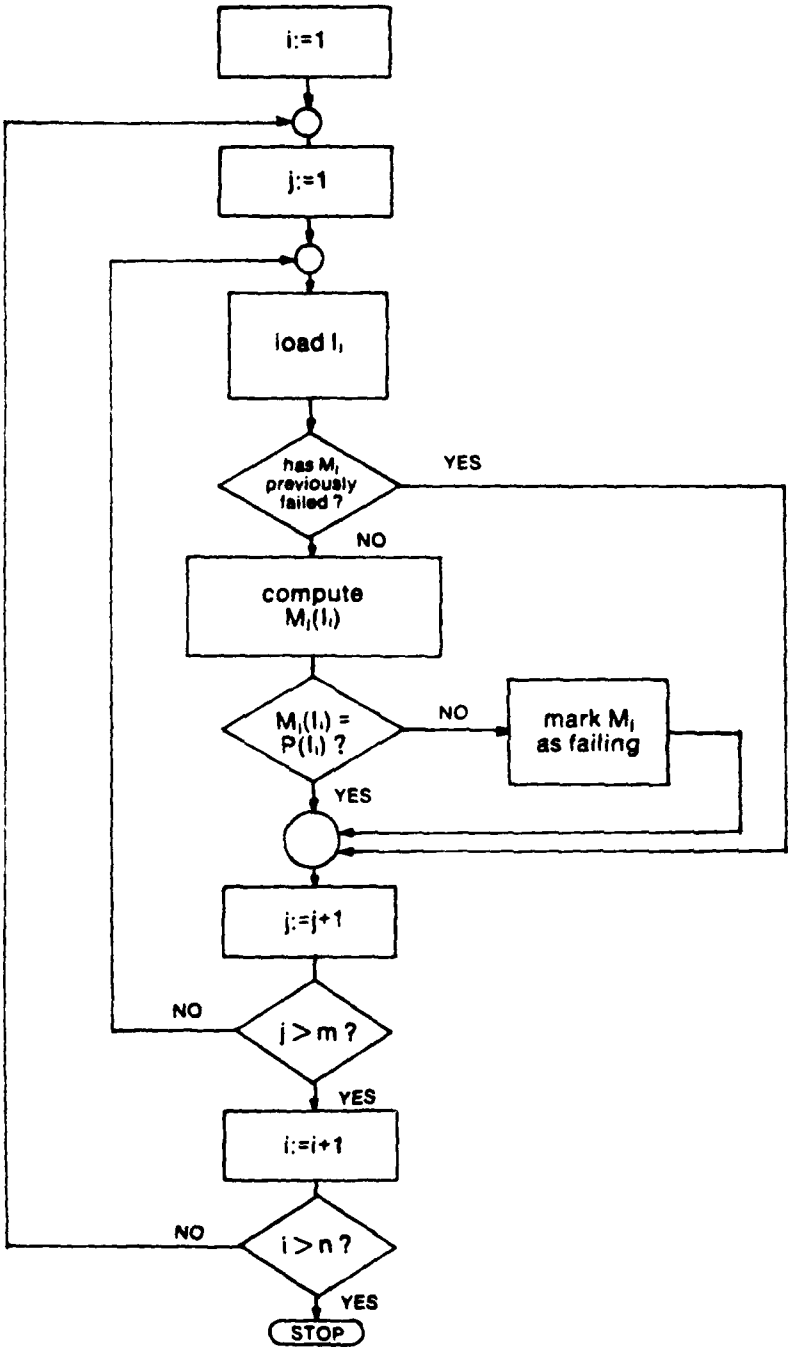


Figure 2-1: Main Loop of Current Program Mutation Systems

by distributed computing. An overview of the system design and an implementation strategy will be given in sections 4 and 5.

2.2. Automatically Detecting Equivalent Mutants

Figure 2-1 illustrates the computational waste of equivalent mutants. Assume that M_j is not equivalent to P and that M_j will fail on some I_i. As soon as M_j fails it is never again executed. However, if M_j is equivalent to P then M_j will be executed on each I_i since M_j can never fail[3].

In a production system virtually all equivalent mutants must be automatically detected for reasons other than efficiency: if equivalent mutants are not accounted for, then the interpretation of the mutant failure percentage can be misleading. Furthermore, it has been observed that manually detecting equivalent mutants can require large amounts of human effort [2, 15], and manual detection of equivalent mutants is an error-prone human activity which again can lead to misinterpretation of the mutant failure percentage.

The design of our prototype distributed system will incorporate the detection method designed in [4]. To experiment on how well the method works and to improve the method one would use a data base of programs with known equivalent mutants as has been collected in [9, 7]. It has been estimated that as many as 95% of the equivalent mutants can be detected automatically [1, 8]. Aside from improving this figure, experiments could suggest strategies to help manually detect the remaining small number of equivalent mutants.

2.3. Partial Mutant Execution

As stated above and illustrated in figure 2-1, the current mutation systems execute mutants in their entirety on test data. In the design of the prototype distributed system we will incorporate an execution time saving feature of commencing mutant execution at the **point of mutation**.

The method can best be illustrated through a straight line program. Assume P is an N statement straight line program with statements $S_1, S_2, \ldots, S_N$. Let T be a test case for P with input I and output O. Let D represent all data variables accessed by P and denote by D_i the state of D after S_i has been executed. Figure 2-2 illustrates this notation. Then if mutant M_j affects statement S_k but doesn't affect statements $S_1, S_2, \ldots, S_{k-1}$ we can, without loss of generality, begin the execution of M_j with data state D_{k-1} at mutated statement S_k.

The major obstacle to be overcome in implementing this speed-up method lies in how to store and

---

[3] An equivalent M_j could fail due to a runtime exception. Experience indicates that this is extremely rare.

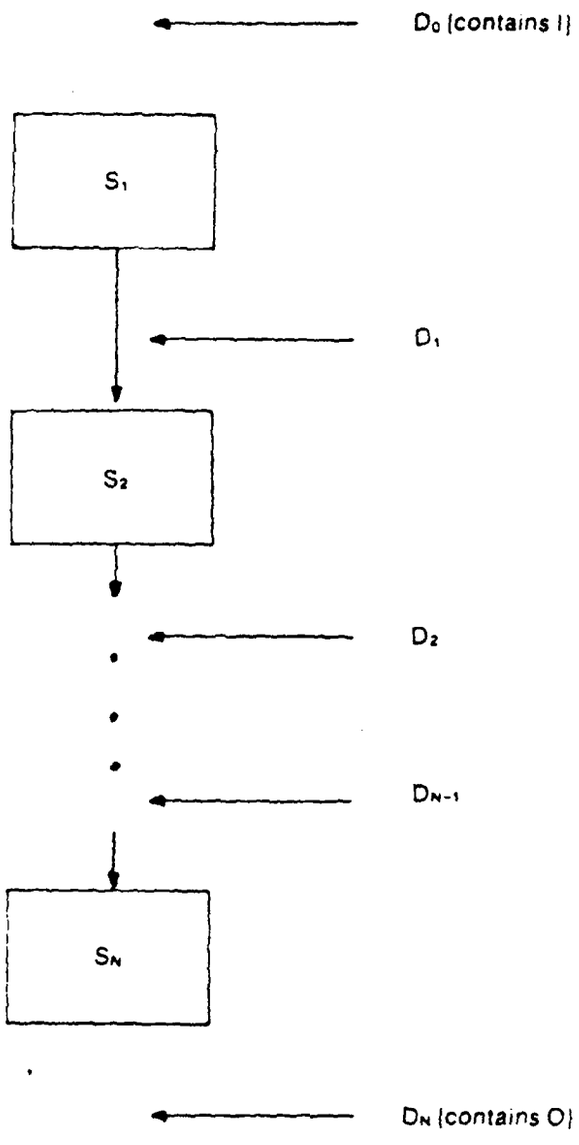


Figure 2-2: Partial Mutant Execution of a Straight Line Program

retrieve P's intermediate data values. The straight-line program example illustrates some issues and ideas to explore in designing the prototype distributed system. Clearly storing $D_0,...D_N$ would work but would require a factor of N additional storage. However, it is easy to see that if the values computed by the S's (rather than the D's) are stored in a reversed linked-list fashion, then D_k can be constructed without executing $S_1,S_2,...,S_k$. This approach can be extended to tree and acyclic graph programs, but breaks down for arbitrary program structures.

It has been shown that a large variety of program structures, usually restricted to intraprocedure analysis, can be reduced to the above three forms [17]. In the prototype distributed program mutation system we will explore using the above techniques at the node level of the reduced flow graph representation of Fortran subroutines. We will also need to construct and use the procedure call graph as defined in [17]. It might be necessary to commence mutant execution in the calling ro'· ıe for test cases which cause subroutines to be called many times.

There is another saving of execution time that can be realized with the above method of commencing mutant execution at the point of mutation. Referring again to figure 2-2, assume mutant M_j affects statement S_k but doesn't affect any other statements.[4] Then we not only can begin execution of M_j at mutated statement S_k but we can, without loss of generality, *terminate* the execution of M_j after executing mutated S_k providing the D_k data state of the mutant matches the D_k data state of the program P -- M_j cannot fail. The case where the data states are unequal will be discussed in the next subsection.

In the detailed design of the distributed mutation system, one should instrument measurement techniques in an effort to estimate how much of a saving is being realized due to partial mutant execution.

2.4. A "Weak" Mutation Option

It was seen above that mutants can be terminated at the point of mutation provided the mutant's data state matches the program's at that point. If they don't match, however, the mutant cannot be terminated and marked as failing since at some later point in its execution its data state could return to match the program's.

It would be unwise to continuously monitor a mutant's data state to see if it has returned to match

---

[4] In the current program mutation systems most, but not all, of the generated mutants have this property

the program since we intuitively feel that such returns are rare.[5] Marking a mutant as failing if it doesn't match the program at the point of mutation will be called *weak program mutation*. Two things are clear about weak program mutation: 1) it can be implemented much more efficiently than program mutation, and 2) it cannot give more information on the correctness of a program than program mutation. What is not clear is how much weaker is weak program mutation than program mutation.

A prototype distributed program mutation system should have weak program mutation as an option. This would allow one to conduct experiments on the above question. The experiments would be of a "beat the system" nature [7] in which a subject takes programs with known errors and tries to develop test data on which the program doesn't fail but on which all mutants of the program fail. Initially, the same programs which have been used in beat the system experiments on program mutation [9] should be used for beat the system experiments on weak program mutation. This will allow a comparison of program mutation and weak program mutation on known results. Later, the improved efficiency of the prototype distributed mutation system would allow one to conduct experiments on programs of much larger size.

3. Design Overview of Current Mutation Systems

The three existing program mutation systems all have the same basic design [6, 5]. There are six major modules:

1. A **parser** -- the program to be tested is parsed into an internal form which is suitable for program mutation.
2. **An Interpreter** -- executes internal form representations of programs and mutants. Detects various run-time failure exceptions.
3. **A Test Case Manager** -- controls execution of the program on the test data and records the output of the program.
4. **A Mutant Generater** -- applies the mutant operators to the program to generate mutant descriptors which indicate what changes to the internal form constitute a mutation.
5. **A Mutant Manager** -- uses the mutant descriptions to create mutants by altering the internal form, controls execution of the mutants on the test data, and maintains tables which indicate the failure status of mutants.
6. **A Report Generater** -- creates a printable summary of the testing run.

From the descriptions of these modules it can be seen that there are four major data structures in

---

[5] Intuitively, if a mutant will return to match the program then it will do so quickly -- within the next few statements. This fits in nicely with the reduced flow graph node-level method of partial mutant execution outlined above.

current mutation systems:

1. The internal form representation of the program,
2. The test cases and the program's output on them,
3. The mutant descriptors, and
4. The mutant status tables.

A testing run on current mutation systems can be broken down into three relatively independent phases:

1. **Phase 1** -- The program is parsed, the program is executed on the test data, and the mutant descriptors are generated.
2. **Phase 2** -- The mutants are executed on the test data.
3. **Phase 3** -- The report is generated.

Figure 3-1 summarizes the design and how testing runs are done on the current program mutation systems in terms of the components described above. Note that in testing a program P, the systems allow the test data to be augmented without redoing what has been previously done in phase 1. Furthermore, the user has the option of applying the mutant operators incrementally from run to run rather than dealing with all mutants from the outset. There are three places indicated in figure 3-1 where a testing run may start:

(A) Start point for the first testing run.

(B) Start point for subsequent runs involving new test data, and possibly the application of more mutant operators.

(C) Start point for subsequent runs involving no new test data but applying more mutant operators.

4. Design Overview of the Prototype Distributed Mutation System

The prototype distributed mutation system will still have the three phases of current mutation systems. Implementing the ideas of section 2 will require some new components as well as major design overhauls to some existing components. There will be two new components:

1. **A Program Flow Graph** -- this data structure will indicate where and how partial mutant execution can be done. It will also indicate the basic blocks of the program.
2. **An Equivalence Tester** -- this module will use the basic block information of the program flow graph to mark mutant descriptors as equivalent.

Five components of existing mutation systems need substantial revisions. They are:

1. **The Parser** -- the program flow graph will be generated by the parser.
2. **The Internal Form** -- the internal form representation of the program will now include the program flow graph.

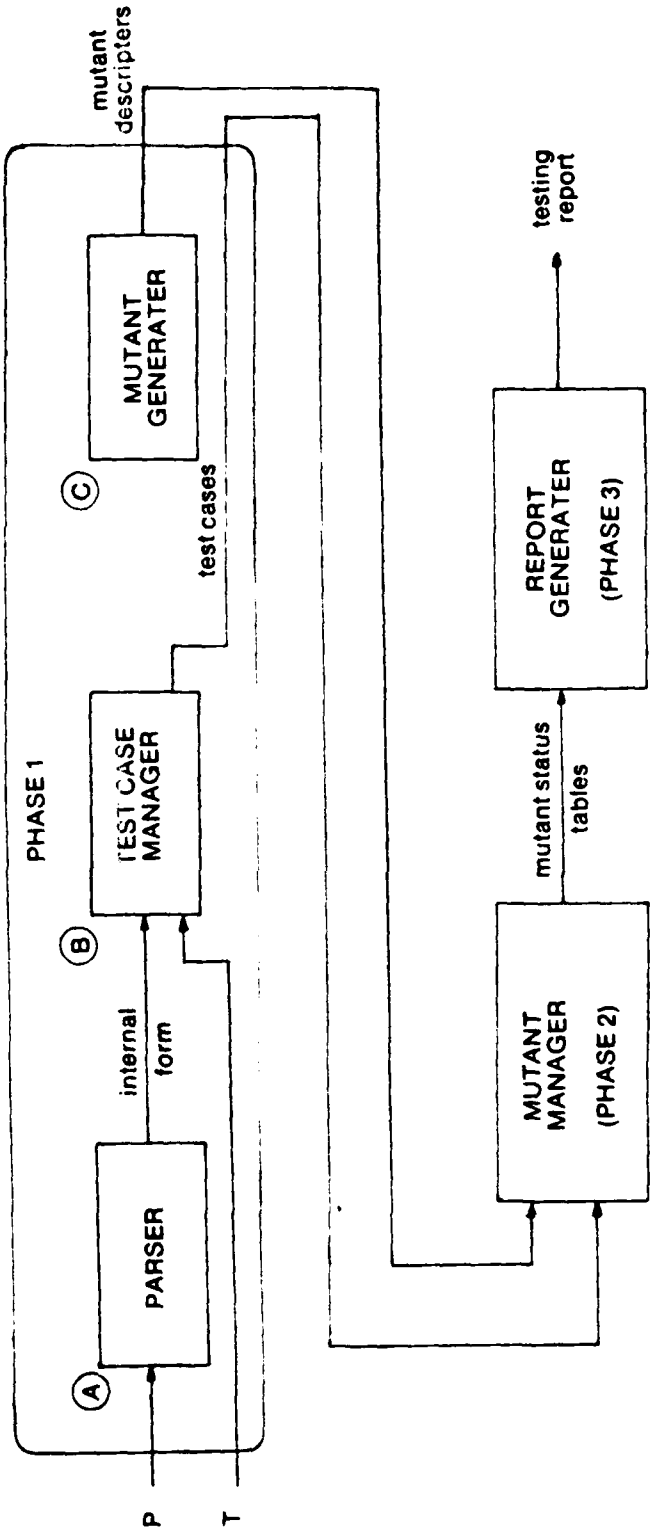


Figure 3-1: Design of Current Program Mutation Systems

3. **The Test Case Manager** -- in order to do partial mutant execution it will be necessary for the test case manager to record data state information on the program at the points indicated by the flow graph.

4. **The Test Cases** -- in addition to the input/output information, the test cases will now contain the intermediate information necessary for partial mutant execution.

5. **The Mutant Manager** -- the mutant manager will now control partial execution of mutants, both in starting mutant execution at the point of mutation and in ending mutant execution immediately thereafter in the case of weak program mutation. In addition, the mutant manager will control the *parallel* execution of mutants. The distributed aspects of the mutant manager will be elaborated below.

The phase 1 design overview of the prototype distributed system is illustrated in figure 4-1. Note that we could have "parallelized" phase 1 to take advantage of a distributed system. We choose to avoid the complexities of doing so because the execution time spent in phase 1 is insignificant with respect to the execution time spent in phase 2.

To describe the distributed aspects of the prototype distributed system, we will use the standard parallel processing abstraction terms of *process*, *message passing*, *father process*, and *son process*, in order to describe the design structure independently of any particular system architecture. An implementation strategy will be described in the next section.

The mutant manager will be a father process capable of creating, managing, and communicating with an arbitrary number of identical son processes called *mutant executers*. The mutant manager will exist for the duration of phase 2 but the mutant executers may come and go. The mutant executers operate independently of each other, don't communicate with each other, and don't know or care about the existance of other mutant executers. The only communications are between the mutant manager and the mutant executers. Figure 4-2 illustrates this process structure.

Upon creation, a mutant executer will contain code for communicating with the mutant manager, the internal form representation of the program, and the interpreter. These components remain resident for the entire existance of a mutant executer. After creation, the mutant manager passes to the mutant executer *one* test case which will be resident in the executer for quite some time. This is done for two reasons: the test case can constitute much data and we wish to minimize communications, and we want the mutant executer to "self-optimize" itself for executing a particular test case and this will be a time consuming activity.

After these two steps the mutant executer is ready to create and execute mutants. For simplicity, this will be done one mutant at a time -- the mutant manager will pass a mutant descriptor to the mutant executer and then wait for a message indicating whether or not the mutant has failed. Note that all communication between the manager and the executer will be of a

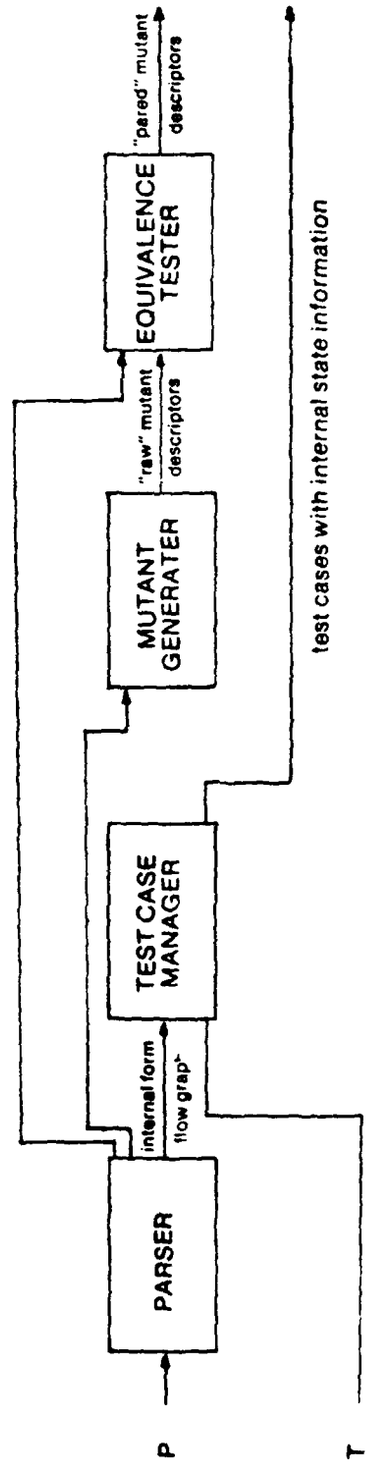


Figure 4-1: Design of the Distributed Mutation System -- Phase 1

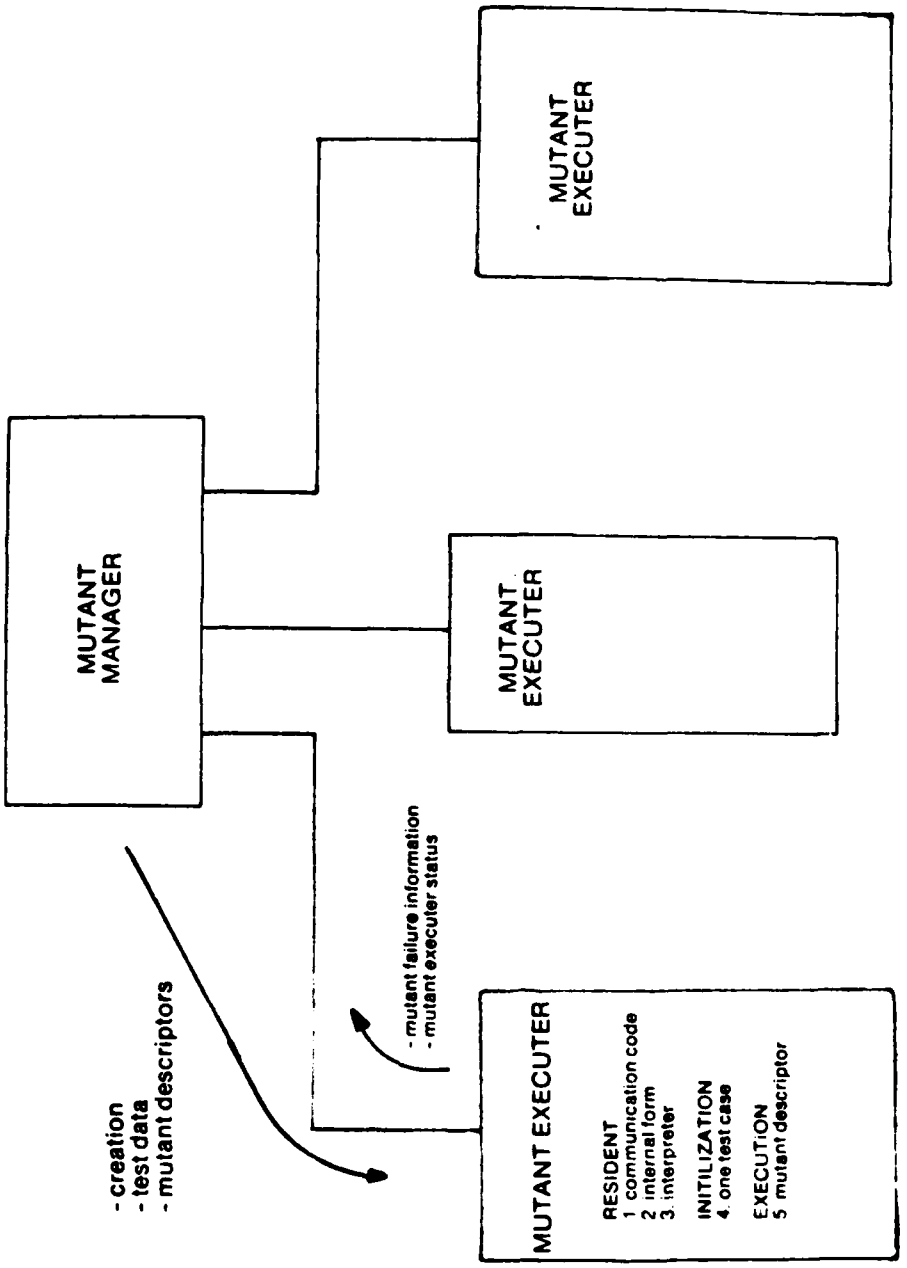


Figure 4-2: Design of the Distributed Mutation System -- Phase 2

command-acknowledgement nature. In addition, the manager can also expect messages from the executer such as "I am about to cease to exist".

5. An Implementation Strategy

The implementation of the prototype distributed mutation system can be done in three successive stages, each building on the former with the first stage building on the current FMS.2 program mutation system. The completion of stage one would permit the performance of the experiments on weak program mutation which were outlined in section 2.

5.1. Stage 1

During this stage FMS.2 would be modified to implement the equivalence tester and partial mutant execution. The necessary changes were summarized in section 4. There will be no introduction of parallelism during stage 1.

5.2. Stage 2

During this stage the mutant manager, the mutant executer, and the communication mechanism between them would be built and the system would evolve toward being of a distributive nature. Depending on the available facilities, it might not be necessary to actually use or simulate distributed hardware during this stage. For example, a DEC-2060 running the TOPS-20 operating system [12] is particularly suited for building the type of distributed system which we have described since it supports a tree-structured hierarchy of asynchronous processes, and interprocess message passing. Thus in stage 2 it is recommended that the prototype distributed mutation system be built on a single processor machine under an operating system which supports in software a realistic version of distributed computation.

5.3. Stage 3

During this stage any artificiality in the distributed mutation system can be removed by converting it to run on a multiprocessor system. There are several such systems currently available. For example, at Yale there are two DEC-2060's connected via an Ethernet-like [20] local area network called Chaosnet [21]. Building the prototype mutation system on this network would allow an examination of the communication issues involved in running mutant executers on different processors.

Unfortunately, for the above network of two DEC-2060's, the mutant manager would be on the same machine as one of the mutant executers. The artificiality that this imposes will be minimal since

the mutant manager will be dormant most of the time. However, even this small degree of artificiality can be avoided if one has available a local area network consisting of many powerful personal computers such as the recently announced Apollo machine [3].

References

[1] Alan T. Acree.
On Mutation.
PhD Thesis, Georgia Institute of Technology, 1980.

[2] A. Acree, T.A. Budd, R. DeMillo, R. Lipton, and F.G. Sayward.
Mutation analysis.
ACM Transactions on Programming Languages and Systems, To appear.

[3] Apollo Computer Systems, Inc.
APOLLO Domain Architecture.
Prelimanary Architecture Notes.
1981.

[4] D. Baldwin and F. Sayward.
Heuristics for determining equivalence of program mutations.
Technical Report 161, Yale University, 1979.

[5] T.A. Budd and F.G. Sayward.
The internal specifications of the pilot mutation system.
Technical Report 118, Yale University, 1977.

[6] T.A. Budd, R.A. DeMillo, R.J. Lipton, and F.G. Sayward.
The design of a prototype mutation system for program testing.
In *Proceedings of the 1978 National Computer Conference*, pages 623-627. AFIPS, 1978.

[7] T.A. Budd, R. DeMillo, R. Lipton, and F.G. Sayward.
Theoretical and experimental studies on using program mutation to test the functional correctness of programs.
In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*, pages 220-233. ACM, 1980.

[8] Timothy A. Budd.
Mutation Analysis of Program Test Data.
PhD Thesis, Yale University, 1980.

[9] T.A. Budd, R. Hess, and F.G. Sayward.
Error Programs and Test Data for Life-Cycle Experiments.
Technical Report 180/80, Yale University, 1980.

[10] Lori A. Clarke.
A System to Generate Test Data and Symbolically Execute Programs.
IEEE Transactions on Software Engineering SE-2(3):215-222, September, 1976.

[11] L.A. Clarke and J.L. Woods.
Program testing using symbolic execution.
Presented at the Navy Laboratory Computing Committee Symposium on Software Specification and Testing Technology.
April 1978.

[12] *TOPS-20 Monitor Calls Reference Guide*, Order #AA-4166-TM.
Digital Equipment Corporation, Marlboro MA, 1980.

[13] R.A. DeMillo, R.J. Lipton and F.G. Sayward.
Hints on test data selection: Help for the practicing programmer.
Computer 11(4), April, 1978.

[14] R.A. DeMillo, R.J. Lipton and F.G. Sayward.
Program mutation: A new approach to program testing.
In *Infotech State of the Art Report: Software Testing*, pages 2(107-128). INFOTECH, 1979.

[15] Jeanne M. Hanks.
Testing Cobol Programs by Mutation.
Master's Thesis, Georgia Institute of Technology, 1980.

[16] Sidney L. Hantler and James C. King.
An Introduction to Proving the Correctness of Programs.
ACM Computing Surveys 8(3):331-353, September, 1976.

[17] M.S. Hecht.
Flow Analysis of Computer Programs.
North-Holland, 1977.

[18] W.E. Howden.
Methodology for the generation of program test data.
IEEE Transactions on Computers (C-24,5), May, 1975.

[19] W.E. Howden.
Reliability of the path analysis testing strategy.
IEEE Transactions on Software Engineering (SE-2,3), September, 1976.

[20] Robert M. Metcalfe and David R. Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
CACM, July, 1976.

[21] David A. Moon.
CHAOSnet.
MIT A.I. Lab Internal Memorandum.
1979.

[22] R. J. Lipton and F. G. Sayward.
The Status of Research on Program Mutation.
In *Digest for the Workshop on Software Testing and Test Documentation*, pages 355-373.
Fort Lauderdale, Florida, December, 1978.

[23] C.V. Ramamoorthy, S.F. Ho and W.T. Chen.
On the automated generation of program test data.
IEEE Transactions on Software Engineering (SE-2,4), December, 1976.


DATE
FILMED
7-8